Generative Representations for Computer-Automated Design Systems

Gregory S. Hornby

Evolvable Systems Group
NASA Ames Research Center, Mail Stop 269-3
Moffett Field, CA 94035-1000
hornby@email.arc.nasa.gov
http://ic.arc.nasa.gov/people/hornby/

Abstract

With the increasing computational power of computers, software design systems are progressing from being tools for architects and designers to express their ideas to tools capable of creating designs under human guidance. One of the main limitations for these computer-automated design programs is the representation with which they encode designs. If the representation cannot encode a certain design, then the design program cannot produce it. Similarly, a poor representation makes some types of designs extremely unlikely to be created. Here we define *generative representations* as those representations which can create and reuse organizational units within a design and argue that reuse is necessary for design systems to scale to more complex and interesting designs. To support our argument we describe GENRE, an evolutionary design program that uses both a generative and a non-generative representation, and compare the results of evolving designs with both types of representations.

1 Introduction

As computers become more powerful, software design tools are becoming increasingly more powerful tools for architects and designers to express their ideas. In addition, the use of artificial intelligence techniques has enabled these software packages to assist in the design process themselves. Already computer automated design systems have been used for the design of antennas, flywheels, load cells, trusses, robots and other structures (Bentley, 1999; Bentley & Corne, 2001). While these programs have been successful at producing simple, albeit novel artifacts, a concern with these systems is how well their search ability will scale to the larger design spaces associated with more complex artifacts. In engineering and software

development, complex artifacts are achieved by exploiting the principles of regularity, modularity, hierarchy and reuse, which can be summarized as the hierarchical reuse of organizational units.

Breaking down a computer-automated design program into its separate modules yields the representation for encoding designs, the algorithm for exploring the space of designs that can be represented, and the fitness function for scoring the goodness of a particular design. Ideally, the ability of the design program to create multiple layers of organizational units should be independent of how designs are scored. In addition, the algorithm for exploring the space of designs can only find designs that can be expressed by the chosen representation. For example, in optimizing the dimensions on a design, the design program can only produce designs that fall in the pre-specified parameter space and no modification of the search algorithm can affect the degree of reuse in the resulting designs. Thus for computer-automated design software to achieve designs with multiple layers of reusable organizational units these software systems must use a representation capable of encoding such designs.

Representations for computer-automated design can be divided into several classes. At the top level representations can be split into *parameterizations* and *open-ended* representations. Parameterizations are the class of representations in which the topology of the design is pre-specified and the search algorithm is performing numerical optimization on a set of parameters, such as the dimensions on a design. In contrast, with open-ended representations the space of design topologies can be explored thereby allowing new types of designs to be discovered. Since we are interested in the ability of a design program to create truly novel designs, we focus on open-ended representations. An important distinction between classes of open-ended representations is whether or not they are generative. With a *generative* representation elements of an encoded design can be used multiple times in mapping from the encoded design to the actual design and with a *non-generative* representation each element in an encoded design is used at most once. For example, with a generative representation the design of a table could encode the specification of a table leg once and then for each occurrence of a table leg there would be a pointer to look at this specification. Using a non-generative representation there would be copies of the specification of the table leg at each place where it is used.

Being able to reuse parts of a design improves the ability of generative representations to scale in complexity and number of parts. In the first case, designs often have dependencies such that changing one component in a design requires the simultaneous change in another component. For example, in creating a design for a dining-room table the length of each table leg is dependent on the lengths of all the other legs in the table and it is only useful to change the lengths of all legs together. By having a single description of a table leg, with references to this description at each place where it is used, all table legs are changed by changing this one description. With a non-generative representation the search program must

find and change all occurrences of a leg together, but this is feasible only when the dependencies are known beforehand and not when they are created during the search process. In the second case, as the number parts in a design increases there is an exponential increase in the size of the design space. Since search consists of iteratively making changes to designs that have already been discovered, this increase in the design space reduces the relative effect of changing a single part in a design and increases the number of changes needed to navigate the design space. Increasing the amount of change made before re-evaluating a design is not a viable solution because this increase produces a corresponding decrease in the probability that the resulting design will be an improvement. With a generative representation the ability to reuse previously discovered assemblies of parts by either adding or removing copies enables large, meaningful movements about the design space. Here the ability to hierarchically create and reuse organizational units acts as a scaling of knowledge through the scaling of the unit of variation.

2 A Generative Representation

Generative representations are defined as any type of representation that allows for the creation and reuse of organizational units in a design. Within this definition there are many different methods by which reuse can be achieved. The generative representation used here is a kind of computer language within which design-constructing programs are written. This language consists of a framework for design construction rules and a set of these rules defines a program for a design. Designs are created by compiling a design program into an assembly procedure of construction commands and then executing this assembly procedure in the module which constructs designs. Since it is a general framework for encoding designs the person using this framework must supply the set of design-construction commands and a design constructor.

The rules for constructing a design consist of a rule head followed by a number of condition-body pairs. For example in the following rule,

$$A(n0, n1) : n1 > 5 \rightarrow B(n1{+}1)cD(n1{+}0.5, n0{-}2)$$

the rule head is A($n0$, $n1$), the condition is $n1 > 5$ and the body is B($n1$+1) c D($n1$+0.5, $n0$-2). A complete encoding of a design consists of a starting command

and a sequence of rules. For example the a design could be encoded as,

$$P0(4)$$

$$P0(n0): \ n0 > 1.0 \rightarrow [\ P1(n0 * 1.5)\ ]\ a(1)\ b(3)\ c(1)\ P0(n0 - 1)$$

$$P1(n0): \ n0 > 1.0 \rightarrow \{\ [\ b(n0)\ ]\ d(1)\ \}(4)$$

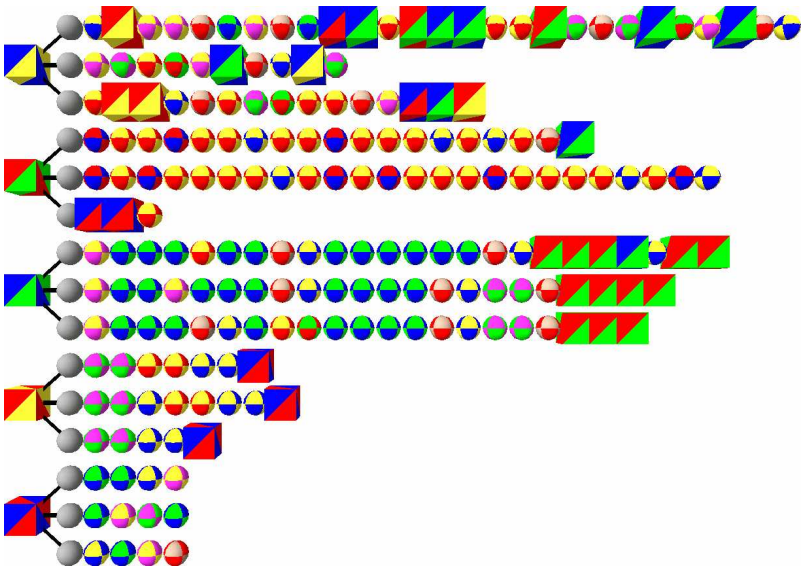
Through an iterative sequence of replacing rule heads with the appropriate body this program compiles as follows,

1. *P0(4)*
2. *[P1(6)] a(1) b(3) c(1) P0(3)*
3. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [P1(4.5)] a(1) b(3) c(1) P0(2)*
4. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [{ [b(4.5)] d(1) }(4)] a(1) b(3) c(1) [P1(3)] a(1) b(3) c(1) P0(1)*
5. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [{ [b(4.5)] d(1) }(4)] a(1) b(3) c(1) [{ [b(3)] d(1) }(4)] a(1) b(3) c(1)*
6. *[[b(6)] d(1) [b(6)] d(1) [b(6)] d(1) [b(6)] d(1)] a(1) b(3) c(1) [[b(4.5)] d(1) [b(4.5)] d(1) [b(4.5)] d(1) [b(4.5)] d(1)] a(1) b(3) c(1) [[b(3)] d(1) [b(3)] d(1) [b(3)] d(1) [b(3)] d(1)] a(1) b(3) c(1) b(3)*

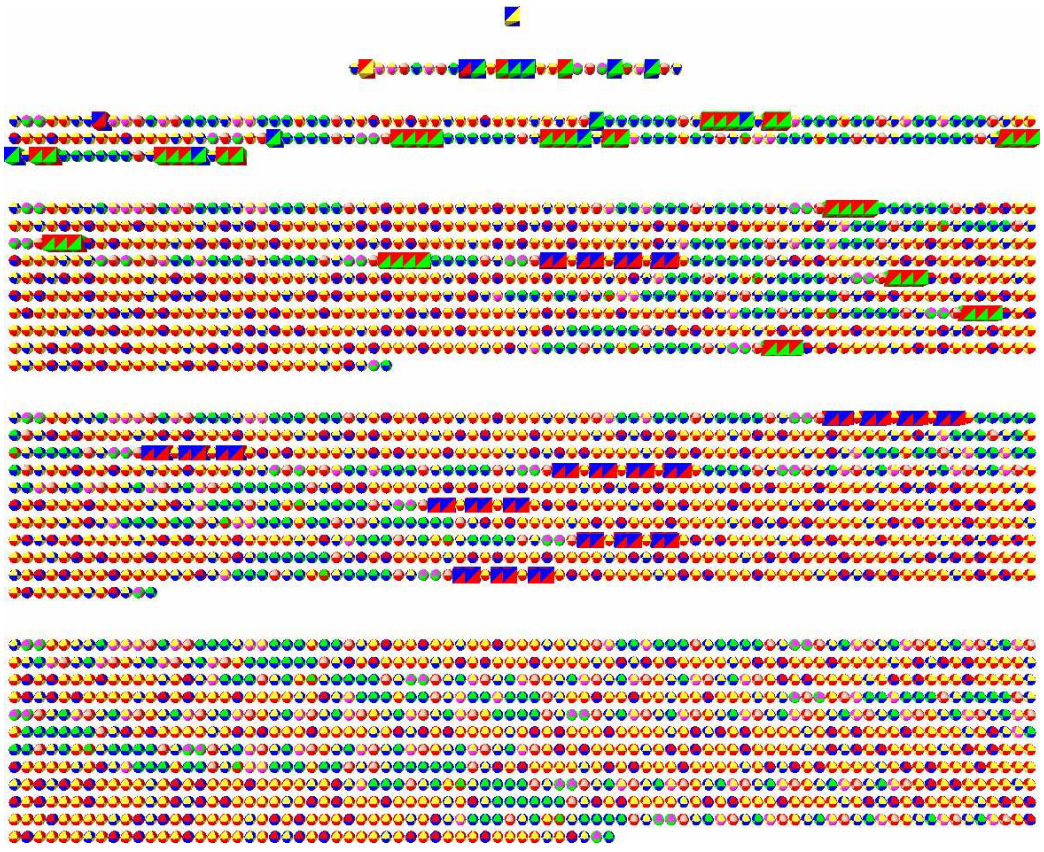
A graphical version of the generative representation described here is shown in figure 1.a, and the sequence of assembly procedures generated by it are shown in figure 1.b. In these images rule-head symbols are represented by cubes with lines connecting them to their condition-body pairs, grey spheres represent the condition and the symbols following it are the body. The sequence is started with the first cube (here a blue and yellow one) and the sequence of symbols below it are the assembly procedures generated after each iteration of parallel replacement.

To create designs with this type of generative representation, the non-rule-head symbols are interpreted as construction commands in a design construction language. For example, three-dimensional objects can be constructed by creating a language for adding cubes in a three-dimensional grid:

- `back`(n), move in the negative X direction n units.
- `clockwise`(n), rotate heading $n \times 90°$ about the X axis.
- `counter-clockwise`(n), rotate heading $n \times -90°$ about the X axis.
- `down`(n), rotate heading $n \times -90°$ about the Z axis.
- `forward`(n), move in the positive X direction n units.
- `left`(n), rotate heading $n \times 90°$ about the Y axis.

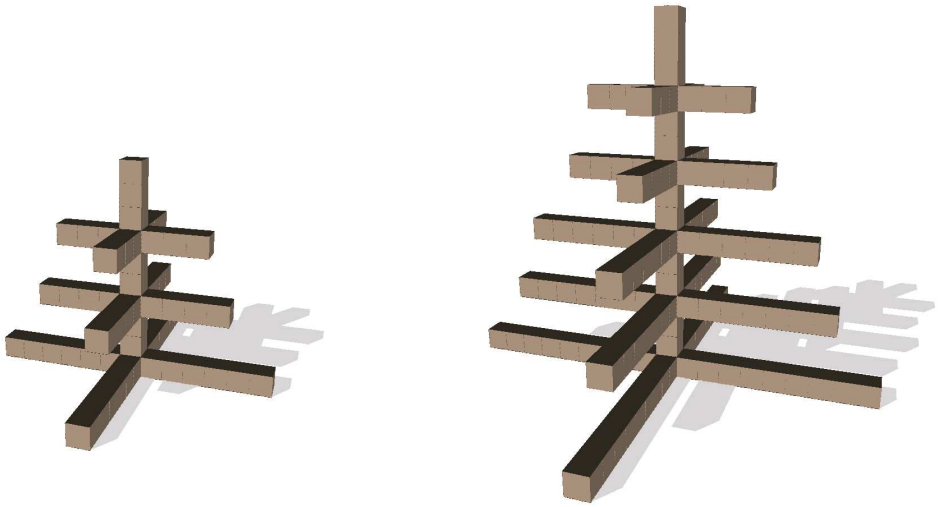


(a)



(b)

Fig. 1. Graphical version of the generative representation, (a); along with the sequence of strings produced, (b).



(a) (b)

Fig. 2. Two tree structures produced from the same set of rules with different starting commands.

- `right`(n), rotate heading $n \times -90°$ about the Y axis.
- `up`(n), rotate heading $n \times 90°$ about the Z axis.
- `]`, pop the top state off the stack and makes it the current state.
- `[`, push the current state to the stack.

With this design-construction language a design starts with a single cube in a three-dimensional grid and new cubes are added with the commands `forward()` and `back()`. The current state, consisting of location and orientation, is maintained and the commands `clockwise()`, `counter-clockwise()`, `down()`, `left()`, `right()`, and `up()` change the orientation. A branching in design construction is achieved through the use of the commands `[` and `]`, which push (save) and pop (restore) the current state onto a stack.

Using the key: a = up, b = forward, c = down, and d = left; the above example becomes,

$$P0(4)$$

$$P0(n0): \; n0 > 1.0 \rightarrow [\; P1(n0 * 1.5)\;]\; up(1)\; forward(3)$$
$$down(1)\; P0(n0 - 1)$$

$$P1(n0): \; n0 > 1.0 \rightarrow \{\; [\; forward(n0)\;]\; left(1)\; \}(4)$$

and compiles into the sequence:

[[forward(6)] left(1) [forward(6)] left(1) [forward(6)] left(1) [forward(6)] left(1)] up(1) forward(3) down(1) [[forward(4.5)] left(1) [forward(4.5)] left(1) [forward(4.5)] left(1) [forward(4.5)] left(1)] up(1) forward(3) down(1) [[forward(3)] left(1) [forward(3)] left(1) [forward(3)] left(1) [forward(3)] left(1)] up(1) forward(3) down(1) forward(3)

Executing this assembly procedure produces the structure shown in figure 2.a. Interestingly, the rules of this design program encode for a family of designs and by using a different starting command different designs can be created. The design in figure 2.b is created by using the starting command $P0(6)$ instead of $P0(4)$.

3 GENRE: An Evolutionary Design System

To demonstrate the advantages of generative representations we use GENRE, an evolutionary design system for creating designs (Hornby, 2003a). GENRE consists of several design constructors and fitness functions, the compiler for the generative representation and an evolutionary algorithm (EA) for searching the design spaces. EAs are an optimization technique inspired by natural evolution (Mitchell, 1996; Michalewicz, 2000) that has been used both in engineering design and also in the creation of different types of art. The EA is the module that drives GENRE and it operates by processing a population of designs (members of which are called *individuals*) encoded with the generative representation. Search is started by creating an initial random population of individuals and evaluating each of these with a user defined *fitness function*, a mathematical expression for scoring the goodness of a design. The EA then creates successive new populations by selecting the better individuals of the current population and applying small amounts of variation to their encoding to produce new individuals in a new population.

The two variation operators that are used to produce new individuals are *mutation* and *recombination*. Mutation creates a new individual by copying the parent individual and making a small change to it, such as by replacing one command with another, perturbing the parameter in a command by adding/subtracting a small value to it, or adding/deleting a sequence of commands in a rule body. Recombination takes two individuals as parents and creates a new individual by making a copy of the first parent and then either exchanging a rule with the second parent, or randomly replacing a sequence of commands in one body with a sequence from the second parent.

Designs can be encoded with either the generative representation described in section 2 or a non-generative representation. For designs encoded with the generative representation the design program is first compiled into a sequence of construction commands called an assembly procedure. This assembly procedure is then executed

by the design constructor to produce the encoded design. For the non-generative representation each individual in the population is an assembly procedure which specifies how to construct the design. We implement this assembly procedure as a degenerate version of the generative representation which has only a single rule in which the condition always succeeds and the body consists only of construction commands. Implementing the non-generative representation in the same way as the generative representation allows us to use the same evolutionary algorithm and the same variation operators with the only difference between the two representations is the ability to hierarchically reuse elements of encoded designs. Once a design has been constructed, using either the generative or non-generative representation, it is evaluated for how good it is with the user-defined fitness function.

4 Evolution of Tables

The first design problem we apply GENRE to is the evolution of tables. The fitness of a table is a product of its height, surface structure, and stability, divided by the number of excess cubes used. The height of a table is the height of the highest cube in the design and the value of a table's surface structure is the number of cubes at this height. Stability is a function of the volume of the table and is calculated by summing the area at each layer of the table. Since maximizing height, surface structure and stability typically result in table designs that are solid volumes, a measure of excess cubes is used to reward designs that use fewer bricks and its value is the number of cubes not on the surface.

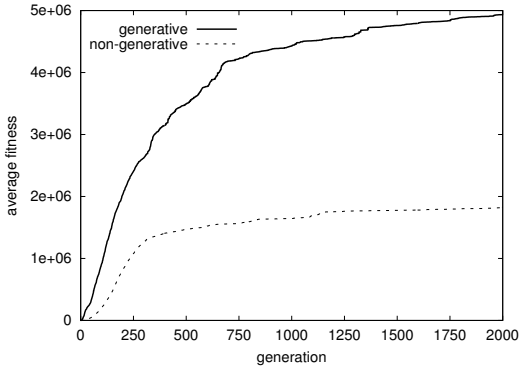


Fig. 3. Fitness comparison between the non-generative and generative representations on evolving tables.

To compare the generative and non-generative representations we ran fifty trials with each representation in a grid of $40 \times 40 \times 40$ cubes. For each trial the evolutionary algorithm was configured to run for two thousand generations with a population of two hundred individuals. The graph in figure 3 contains the results of these experiments. Evolution with the generative representation increased in fitness faster than with the non-generative representation and had a higher final average fitness

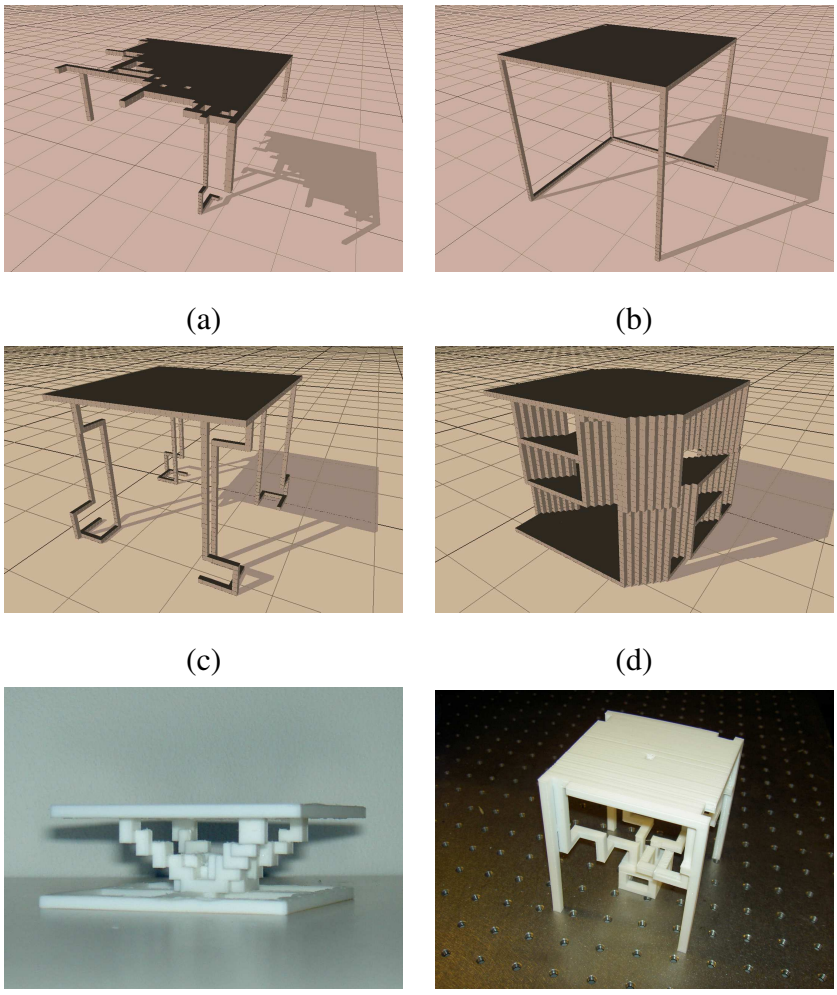


Fig. 4. Evolved tables: (a) the best table evolved using the non-generative representation; (b) the best table evolved using the generative representation; (c)-(f) are other tables evolved using the generative representation with variations of the original fitness function.

of approximately five hundred thousand versus a final average of just under two hundred thousand with the non-generative representation. In addition, the greater leveling off off the fitness curve with the non-generative representation suggests that it does not handle increased design complexity as well as the generative representation.

Images of tables evolved with the two representations show the different styles achieved with them. Examples of the best table evolved with each representation, along with additional tables evolved with the generative representation, are shown in figure 4. The number of parts in these tables range from under a thousand to 5921 for the table in figure 4.d (created with a generative representation). In general, tables evolved with the non-generative representation are irregular and evolution with this representation tends to produce designs in which tables are supported

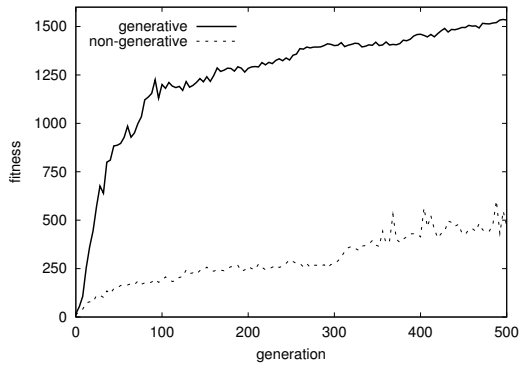


Fig. 5. Performance comparison between the non-generative representation and the generative representation on evolving robots.

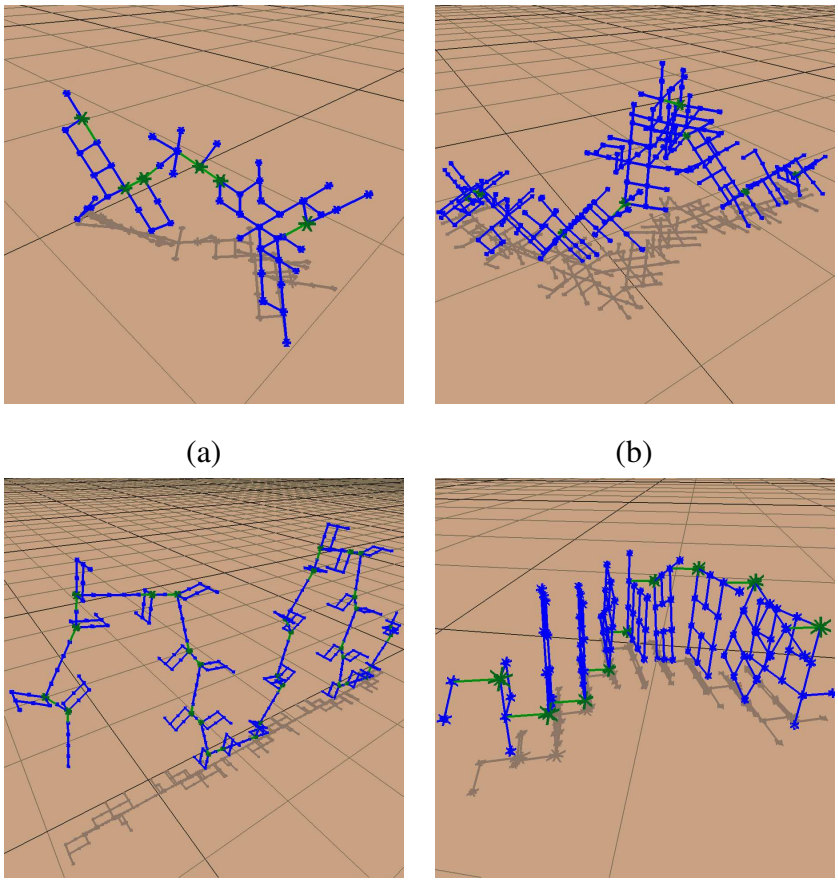
by only one leg. The likely reason for this is that it is not possible to change the length of multiple table-legs simultaneously with the non-generative representation, so the best designs (those with the highest fitness) had only one leg that raised the surface to the maximum height. In contrast, tables evolved with the generative representation have a reuse of parts and assemblies of parts and are supported with multiple-legs.

5 Evolution of Robots

The next class of design substrates on which the non-generative and generative representations are compared is that of designing robots in a simulated, three-dimensional environment. Robots (called *genobots* for ones created with the generative representation) are constructed in a method that is similar to the that of building tables although instead of working with cubes the commands in this design substrate specify the attachment of rods and actuated joints. In addition to constructing the morphology of a robot, there are additional commands which specify the software for controlling the robot. Since the goal for this class of designs is to produce robots that move quickly, a robot's fitness is a function of how far it moved its center of mass.

Ten trials were performed with each representation and figure 5 contains a plot of the averaged best fitness for these trials. After ten generations the generative representation achieved a higher average than runs with the non-generative representation do after 250 generations and the final genobots evolved with the generative representation are on average more than ten times faster than robots evolved with the non-generative representation.

Figure 6 shows the best robot evolved with each representation. From these images it can be seen that the robot evolved with the non-generative representation is more irregular than genobots evolved with the generative representation. In addition, in



(a) (b)

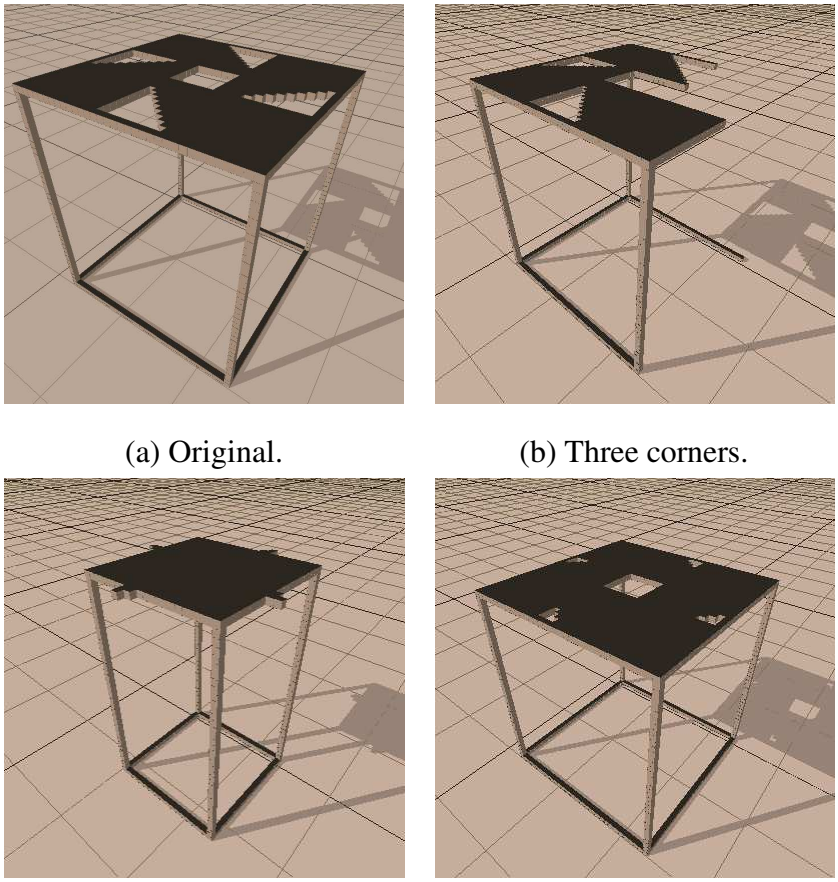
(c) (d)

Fig. 6. Evolved robots: (a) the best robot evolved using the non-generative representation; (b) the best robot evolved using the generative representation; and (c) and (d) are results from additional runs using the generative representation.

some cases genobots encoded with the generative representation have two or more levels of reused assemblies of components. Additional runs with the non-generative representation failed to yield any designs better than the one already shown. In contrast, additional runs with the generative representation produced a variety of genobots with different styles of locomotion, two of which are shown in figure 6.

6 Advantages of a Generative Representation

The central claim of this paper is that using generative representations improves the evolvability of designs by capturing design dependencies and improving the ability of the search algorithm to navigate through large design spaces in a meaningful way. This can be intuitively understood by looking at some examples of designs evolved with a generative representation.



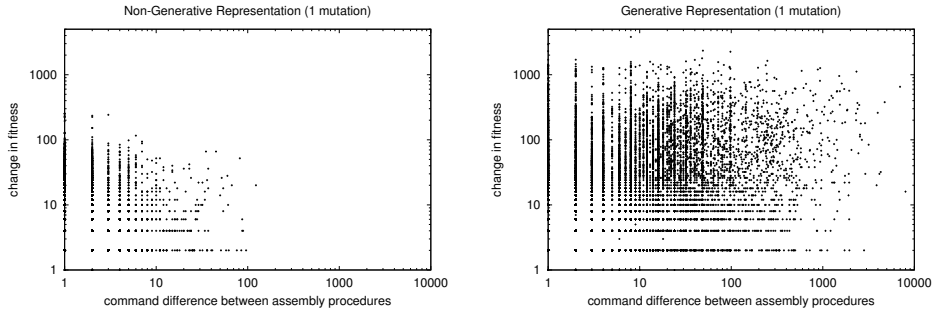
(a) Original. (b) Three corners.

(c) Narrower. (d) More surface cubes.

Fig. 7. Mutations of a table.

Figure 7 contains examples of different tables that can be produced with a single change to an encoded design. The original table is shown in figure 7.a and one change to its generative encoding can produce a table with: (b), three legs instead of four; (c), a narrower frame; or (d), more cubes on the surface. With a non-generative representation these changes would require the simultaneous change of multiple symbols in the encoding. Some of these changes must be done simultaneously for the resulting design to be viable, such as changing the height of the table legs, and so these changes are not evolvable with a non-generative representation. Others, such as the number of cubes on the surface, are viable with a series of single-voxel changes. Yet, in the general case this would result in a significantly slower search speed in comparison with a single change to a table encoded with a generative representation.

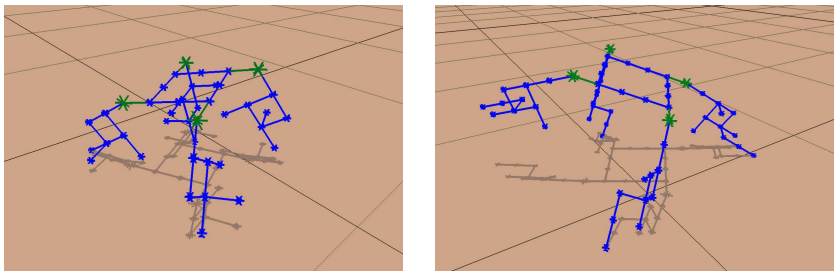
The graphs in figure 8 are scatter plots of the command difference between a parent and child's assembly procedures against their change in fitness on the robot design problem. These graphs show that as the size of change in the resulting design increases it is more likely to be an improvement on designs encoded with a generative representation than those encoded with a non-generative representation. This



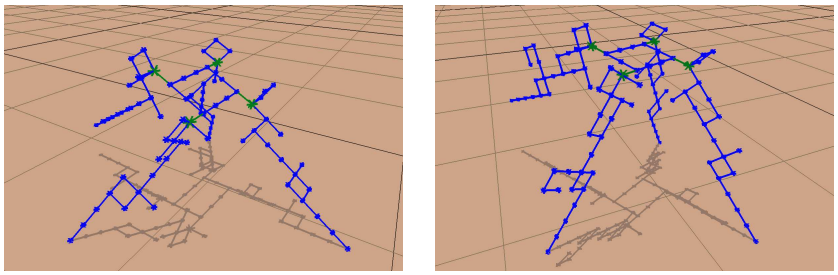
(a) (b)

Fig. 8. Plot of amount of change in assembly procedures from parent to child versus change in fitness for trials evolving robots.

means that search algorithms are better able to use large movements in the design space to navigate through the design space with the generative representation.



(a) Fitness: 348. (b) Fitness: 780.



(c) Fitness: 1450. (d) Fitness: 2192.

Fig. 9. Evolution of a four-legged walking genobot.

That the evolutionary design system is taking advantage of the ability to make coordinated changes with a generative representation is demonstrated by individuals taken from different generations of the evolutionary process. The sequence of images in figure 9, which are of the best individual in the population taken from different generations, show two changes occurring. First, the rectangle that forms the body of the genobot goes from two-by-two (figure 9.a), to three-by-three (figure 9.b), before settling on two-by-three (figures 9.c-d). These changes are possible with a single change on a generative representation but cannot be done with a single

(a) (b)

Fig. 10. Two tables from a family of designs.

change on a non-generative representation. The second change is the evolution of the genobot's legs. That all four legs are the same in all four images strongly suggests that the same module in the encoding is being used to create them. As with the body, changing all four legs simultaneously can be done easily with the generative representation by changing the one module that constructs them, but would require simultaneously making the same change to all four occurrences of the leg assembly procedure in the non-generative representation.

One other advantage of using a generative representation is that by encoding an object through a set of reusable rules for constructing it, it is possible to encode a class of designs. The example encoding of section 2 can produce a family of tree-like structures by changing the argument to the starting command. By evaluating an individual with different parameters to its starting command families of designs can be evolved, such as the tables in figure 10 (Hornby, 2003b).

7 Conclusion

The purpose of this work has been to argue that for computer-automated design systems to scale in complexity they must use generative representations: representations which allow for the hierarchical construction of reusable organizational units. To support this claim we described a generative representation and GENRE, an evolutionary design system for evolving different classes of designs. Using GENRE we showed that evolution with a generative representation produces table and robot designs with a higher fitness than with a non-generative representation. This fitness advantage is a result of the generative representation capturing some design dependencies through reuse and because the ability to manipulate the reusable organizational units of a generative representation better enables the search algorithm to navigate through the design space.

The designs that we can achieve are limited only by our imagination and the tools

with which we work. Similarly, the designs that computer-automated design software can achieve are limited only by the representations with which they operate. The generative representation described in this paper is just one way of allowing organizational units to be reused in a design. There are many alternatives, such as variations of cellular automata, models of developmental biology, as well as actual computer programs, each with its own strengths and weaknesses. For now it is premature to say which direction is best, but as representations become increasingly more powerful in hierarchically encoding organizational units so too will computer-automated design systems improve in their ability to produce ever more complex and interesting designs.

References

Bentley, P. J. (ed). 1999. *Evolutionary Design by Computers*. San Francisco: Morgan Kaufmann.

Bentley, P. J., & Corne, D. W. (eds). 2001. *Creative Evolutionary Systems*. San Francisco: Morgan Kaufmann.

Hornby, G. S. 2003a. *Generative Representations for Evolutionary Design Automation*. Ph.D. thesis, Michtom School of Computer Science, Brandeis University, Waltham, MA.

Hornby, G. S. 2003b. Generative Representations for Evolving Families of Designs. *Pages 1678–1689 of:* et al., E. Cantu-Paz (ed), *Proc. of the Genetic and Evolutionary Computation Conference*. LNCS 2724. Berlin: Springer-Verlag.

Michalewicz, Z. 2000. *Genetic Algorithms + Data Structures = Evolution Programs*. Third edn. Berlin: Springer-Verlag.

Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press.